AUVic Communication System

Revision 1.1

By
Donovan Parks

Date
February 2, 2003

TABLE OF CONTENTS

0.0 Introduction

The AUVic Communication System (AUVic-CS) allows for communication between the host and all systems on the AUV. Communication with all systems is done through a well-defined interface enforced by the AUVic-CS. Systems are effectively “black boxed” as only the communication protocol needs to be known to interact with the system and systems are inherently independent of each other.

The AUVic-CS consists of three types of components: the host PC, the hub, and intelligent peripherals (sensor or actuator). Each intelligent peripheral is a self contain unit that can be communicated with using the protocol defined in this documentation. An intelligent peripheral is required to follow this protocol and will be independent of any other systems on the AUV (except for power). This allows these peripheral to be designed in parallel.

The hub acts as a protocol translator between the host and intelligent peripheral network. The intelligent peripherals communicate with the hub over a multi-drop, half-duplex Inter-Integrated Circuit (I2C) network and the host communication with the hub via a standard RS232 serial port. The hub also provides error checking and recovery and has the ability to reset any peripheral on the I2C network by toggling the Dallas 1-Wire (D1W) digital switch contained on each peripheral.

A common backplane containing power and signal lines is routed to each peripheral and the hub. The power lines provide each peripheral with +5V, +12V, -12V, and ground. The signal lines consist of both the I2C and D1W network lines and an end mission signal line. The end mission line is controlled by an external switch on the AUV and informs all systems when a mission run has been completed or aborted.

The host performs integrated processing of data received from the various intelligent peripherals in the AUV. This document will only refer to the operation of the host in so far as it is involved in the communication system. For detailed information on the operation of the host please refer to the documentation AUVic Host Operation (in progress).

This document discusses the AUVic-CS in detail so that future AUVic teams can understand its operation, enhance the design if required, and troubleshoot the system should problems arise.

1.0 Specifications

All intelligent peripheral must communicate via I2C using open collector pins. The hub circuit board contains the required pull-up resistors to allow the I2C bus to float high. The I2C bus uses a regulated +5V to indicate a high.

ELECTRICAL CHARACTERISTICS FOR HUB

Symbol	**Parameter**	**Conditions**	**Value (Typical/Max)**	**Units**
V_{HUB}	Hub Supply Voltage		5 / -	V
I_{HUB}	Supply Current for Hub (Note 1)			mA

SYSTEM CHARACTERICTICS
(Pull-up Resistors on I2C Lines, R_{UP} = 4.7k, V_{DD} = +5V)

Symbol	**Parameter**	**Conditions**	**Value (Min/Typical/Max)**	**Units**
L_{I2C}	I2C Operating Length[1]	22 AWG solid wire	3.5 / - / -	m
L_{D1W}	D1W Operating Length[2]	22 AWG solid wire	3.5 / - / -	m
L_{SERIAL}	Serial Operating Length	Standard serial cable	3.0 / - / -	m
N_{SLAVES}	Number of Slaves[3]		0 / - / 112	
B_{I2C}	I2C Baudrate[4]		100/400/1000	Kbps
B_{SERIAL}	Serial Baudrate[4]		300/115/921	Kbps

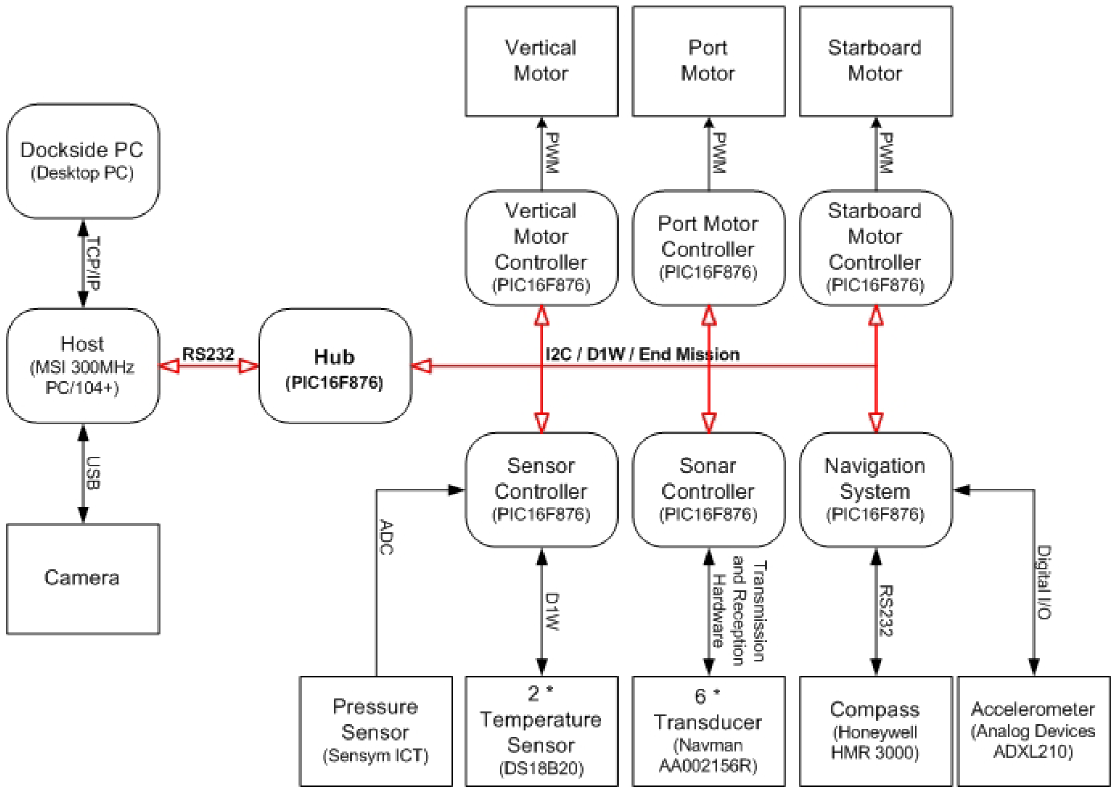
[1] Using a simple pull-up resistor on the I2C lines. Length can be increased by using a current amplified [1].
[2] Using a simple pull-up resistor on the D1W line.
[3] Maximum number of slaves may be limited by the required electrical specification of the I2C bus.
[4] System has only been tested at typical baudrate settings.

PROGRAMMABLE VALUES (Can be set at design time by modifying the Hub program)

Symbol	**Parameter**	**Nominal Value**	**Units**
TO_{I2C}	I2C Timeout	3	ms
TO_{SCI}	SCI Timeout	6	ms
$RETRIES_{SLAVE}$	Times Hub will try to communicate with a slave on the I2C network before considering it offline	2	
$RETRIES_{HUB}$	Times Host will try to communicate with Host before considering a packet faulty	2	

2.0 Infrastructure

The communication system allows communication between the host and all other system on the AUV (with the exception of the camera that is attached via USB). Figure 3.1 provides an overview of the systems on the AUV and illustrates how the AUVic-CS is used to allow communication with different sensors and actuators. The sections specific to the AUVic-CS are shown in red. The hub acts as a serial (RS232) to I2C protocol translator and provides error detection and recovery to improve system performance (see section 8 for details).



Vertical Motor
Port Motor
Starboard Motor
PWM
PWM
PWM
Dockside PC (Desktop PC)
TCP/IP
Vertical Motor Controller (PIC16F876)
Port Motor Controller (PIC16F876)
Starboard Motor Controller (PIC16F876)
Host (MSI 300MHz PC/104+)
RS232
Hub (PIC16F876)
I2C / D1W / End Mission
USB
Camera
Sensor Controller (PIC16F876)
Sonar Controller (PIC16F876)
Navigation System (PIC16F876)
ADC
D1W
Transmission and Reception Hardware
RS232
Digital I/O
Pressure Sensor (Sensym ICT)
2 * Temperature Sensor (DS18B20)
6 * Transducer (Navman AA002156R)
Compass (Honeywell HMR 3000)
Accelerometer (Analog Devices ADXL210)

Figure 2.1 Communication System Overview

2.1 RS232 Connection

A standard serial cable connects the host to the hub. This connection has been tested with a 3m long serial cable, as this length should easily be sufficient for the AUV. The serial link is capable of operation at speeds between 300bps to 921Kbps, but the AUVic-CS has only been tested at 115Kbps. Higher speeds may require the interrupt service routine (ISR) in the hub to be implemented in assembler.

2.2 I2C Network

The Inter-Integrated Circuit (I2C) protocol was developed by Philips Inc. for the high speed transfer of data between IC in close proximity to each other. An I2C network consists of a data line (SDA) and a clock line (SCL). Both lines are open-collector and require a pull-up resistor. An I2C network may be setup in a single master, multiple slaves or a multiple master configuration.

The current master is responsible for driving the clock line and has complete control over the network. Slave addresses can be set to 7 or 10-bits

The AUVic-CS uses 7-bit addressing in a single master, multiple slave configuration. With 7-bit address the network support 112 usable addresses (several address are reserve for special operation), but the electrical specification of the I2C network are the limiting factor on how many peripherals can be supported. The network has been tested to a length of 3.5m using 22 AWG solid wires and 4.7Kohm pull-ups on each line. This length should be sufficient for the AUV and greater lengths have not been tested. The hub is capable of operating a 100Kbps, 400Kbps, or 1 Mbps I2C network, but the AUVic-CS has only been tested at 400Kbps.

An I2C network is a master/slave network so it is common to refer to the master of the network and the slaves. In the AUVic-CS the master is the hub and the slaves are the peripherals. On occasion, this document will use this terminology when referring to the hub (master) or peripherals (slaves).

2.3 Dallas 1-Wire Network

The D1W network supports 2^48 logical addresses, but the electrical specifications of the D1W network are the limiting factor on how many devices can be supported. The D1W network has been tested to a length of 3.5m using 22 AWG solid wires and 4.7Kohm pull-ups. This length should be sufficient for the AUV and greater lengths have not been tested. The hub performs D1W communication in software and the baud rate has not been calculated as it is not critical. For information on how the D1W network is used in the AUVic-CS see section 4.

2.4 Common Network Lines (Backplane)

Several power lines and signals are common to the nodes in the AUVic-CS. All nodes will receive 5V, 12V, –12V, and ground from a common power supply in order to operate. The hub and all peripherals will contain the I2C network signals (SCL and SDA), the D1W signal, and an end mission signal (see section 7.0). Additional control lines are also being carried to each board for future expansion and are connected to I/O pins on the hub.

2.5 Dockside PC (TCP/IP Network)

As shown in Figure 2.1, a dockside PC will also be able to communication with the AUVic-CS via a TCP/IP connection to the host. This will allow manual control of the AUV, the ability to log data, and greatly aid in debugging. For more information please consult “AUVic TCP/IP Network” (in progress).

3.0 Dallas 1-Wire Network

The Dallas 1-Wire (D1W) network allows the hub to reset any peripheral on the I2C network. Each peripheral contains a DS2406 D1W digital switch IC that can reset a peripheral when the hub toggles the switch. *It is a requirement of the communication protocol that all peripherals on the I2C network provide this capability.* This allows the hub to reset a system if it is hanging the I2C network.

Each DS2406 IC has a unique address that is burned into it at the factor. This address is guaranteed to be unique and allows multiple DS2406's to reside on a D1W bus. During design time this address is read from the device (see Appendix V for example code) and related to the I2C address of the peripheral by the host. The host can inform the hub to reset a given peripheral if it is believed to have stopped operating correctly (sections 8 and 9 for how the hub and host determines if a system should be reset).

4.0 End Mission Signal

The end mission signal is contained on the backplane running between the hub and all peripherals in the AUV. This signal informs all peripheral that a mission run is over and is triggered by toggling an external switch on the AUV. The AUVSI competition rules require that all systems cease processing and collecting data when a mission run has ended and that all actuators (i.e. the motors) move to an idle state. It is recommended that this signal be attached to an interrupt pin to ensure a peripheral promptly responds to the signal.

5.0 <u>Protocol</u>

5.1 <u>Flow of Packets</u>

The AUVic-CS operates as a half-duplex network with strict rules controlling the flow of packets. The rules governing the AUVic-CS are:

1. All communication starts with a packet sent by the host.
2. The host requires a reply to every packet.

These two rules are sufficient to describe the operation of the AUVic-CS. Figure 5.1 illustrates how a typical packet moves through the network. The host sends a packet to the hub via RS232. If the packet is addressed to the hub than it immediately processes the packet and sends a reply to the host. The host is now free to send another packet. If the packet is for a peripheral on the I2C network, then the hub will simply forward the packet as it is received from the host. It is a requirement of all peripherals that they send a reply to every packet they receive. The hub forwards any data it received on the I2C network to the host. The host is free to send another packet only after it has received a reply.

These rules imply several invariants about the AUVic-CS:

- a peripheral can only receive a packet from the host (forwarded by the hub)
- a peripheral can only send a packet that is a reply to the host
- packets addressed to the hub come from the host
- the hub can only send a packet that is a reply to the host
- a reply must be sent for every packet received
- the host can only receive a packet that is a reply

Of course, it is possible that the host will not receive a reply to a packet due to a corrupt packet. Sections 7, 8, and 9 discusses in detail how the AUVic-CS detects and handles errors that can occur.

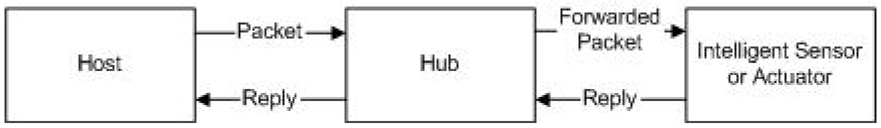


Figure 5.1. Typical Network Operation

5.2 <u>Packet Structure</u>

The AUVic-CS uses packets to send information between the host, hub, and peripherals on the I2C network. Figure 5.2 shows the structure of these packets.

Destination \ Source (1 Byte)
Length (1 Byte)
Type (1 Byte)
Contents (0-20 Bytes)
Checksum (2 Byte)

Figure 5.2 Packet Structure

Each node on the I2C network along with the host is given a unique address. The destination/source byte is used to send packets to a specific node on the network. The half-duplex nature of the AUVic-CS, along with the fact that all packets originate from the host, is taken advantage of to allow a single byte to represent either the destination or source of a packet. This allows the host to check who a reply packet is from. This is useful as the host may send a packet to a node of the I2C network, but receive a reply from the hub indicating that an error has occurred.

When a packet is traveling from the host to the hub or a node on the I2C network the destination/source byte represents the intended destination of the packet. When a packet is traveling from the hub or a node on the I2C network the destination/source byte represents the source of the reply packet. For example, if the host sends a packet to the port motor controller then the destination\source is set to the port motor controller address. The reply from the port motor controller also sets the destination\source byte to the port motor controller address to indicate it is the source of the reply. There is no need to indicate that the reply is for the host as replies are *always* for the host.

The length byte indicates the number of content bytes in a packet. The minimum length of a packet is 0 (destination/source, length, type, 0 content bytes, checksum) and the maximum length of a packet is set to 20 (destination/source, length, type, 20 content bytes, checksum). The length byte is used by nodes on the network to determine when a complete packet is received.

The type byte informs a node on the network what sort of packet it has received. Every packet that a node can receive must have a unique type byte so it knows how to correctly process the packet. For example, the motor controllers can accept over 15 different packet types (set motor, set acceleration, set current limit, etc...) and must perform a different action for each type of packet.

A packet can contain between 0 and 20 content bytes. How the content bytes are interpreted is dependent on the packet type. For example, when a motor controller received a set motor packet it interprets the 2 content bytes as a 16-bit word where the MSBit indicates the direction and the 10 LSBits represents the motor speed between 0 (stop) and 1023 (full speed). It is the responsibility of each peripheral in the system to define what packet types it can accept and the format of any content bytes in these packets.

The checksum is used to verify that a packet has been received without error. The AUVic-MC uses a 16-bit CRC checksum based on the CCITT polynomial ($x^{16}+x^{12}+x^5+1$). [2] can be consulted for more information on how a CRC checksum is generated. When a node receives a packet it calculates the expected checksum and compares it to the received checksum. If the two checksum do not agree than the node can conclude that the received packet contains errors (see sections 7, 8, and 9 for details on how corrupt packets are handled).

It was deemed that a 16-bit CRC checksum provided the best mix of characteristics (see Table 5.1). The increased probability of receiving a corrupt packet that passes the CRC check – an undetected communication error – from using an 8-bit CRC is too high. A 32-bit CRC checksum requires too much program memory and causes an unacceptably low level of packet efficiency. The AUVic-CS consists of relatively short line lengths in an electrically quite environment so the number of packets that will become corrupt should be minimal and the AUVic-CS is capable of recovering from corrupt packets (see section 8 and 9).

Characteristic	**8-bit CRC**	**16-bit CRC**	**32-bit CRC**
Memory (bytes)	256	512	1024
Approx. Processing Time (per byte)[1]	400 ns	1000 ns	2000 ns
Worst-case Burst Error Detection (%)	99.219	99.997	99.999

[1] Assuming a PIC16F876 using a 20MHz clock

Table 5.1 Comparison of CRC Table Look-up Algorithms

5.3 Standard Packets

The AUVic-CS defines several standard packets that all peripherals on the I2C network and the hub must be able to respond to. These standard packets are network services that provide the host with a uniform way to handle commonly required tasks. Each of these network services (Acknowledge, Error, Diagnostic, Status, End Mission, Shutdown) along with broadcasting will be discussed.

An acknowledge packet provides nodes with a uniform way to reply to a packet. The reply to all packets should be of this type unless otherwise specified. Having a uniform packet type for replies simplifies the logic required to process replies. Figure 4.4 gives the structure for an acknowledge packet.

Destination (1 Byte)
Length (1 Byte)
Type (10)
Contents (Variable)
Checksum (2 Byte)

Figure 4.4 Acknowledge Packet

Error packets provide a uniform way in which nodes can report an error to the host. A single content byte is used to indicate the type of error. Appendix V contains a listing of predefined error types that is very comprehensive, but a peripheral is free to define other error codes if required. For example, if a node determines that a packet is corrupt because the checksum check fails then it can respond with an error packet with the content byte set to SLAVE_CRC_FAILED. Figure 4.5 gives the structure for an error packet.

Destination (1 Byte)
Length (0)
Type (15)
Error Code (1 Byte)
Checksum (2 Byte)

Figure 4.5 Error Packet

The diagnostic packet is used by the host to determine if a given address is present on the system. During initialization the host can send out diagnostic packets to determine what peripherals are currently attached to the network. The host can indicate an error if a given peripheral is not present. Ideally, the host would then try to accomplish its mission with the peripherals that are currently available. In practice, it may be more economical to simply resolve the error. The reply to a diagnostic packet is a diagnostic packet and not an acknowledge packet in order to allow the host to take special action for diagnostic packet if desired. The structure of the diagnostic packet is given in Figure 4.6.

Destination (1 Byte)
Length (0)
Type (1)
Checksum (2 Byte)

Figure 4.6 Diagnostic Packet

A status packet is used to query a peripheral for its current state. The response to this packet is dependent on the peripheral, but should contain the value of all properties that can be modified by "set" packets. That is, any property of a peripheral that can be modified by sending it a packet should have its current value contained in the reply to a status packet. For example, the motor controllers implement a "set temperature limit" packet so will provide the current temperature limit setting in the reply to a status packet. The number of content bytes and how they are interpreted is defined by each peripheral. Figure 4.7 gives the structure of a status packet.

Destination (1 Byte)
Length (1 Byte)
Type (3)
Contents (Variable)
Checksum (2 Byte)

Figure 4.7 Status Packet

The end mission packet informs a peripheral that a mission run is over. At the end of a mission all sensor must stop collecting data and all actuators should be brought to an idle state. The end mission packet allows the host to determine that a mission should be considered complete. During a competition run the end mission signal will be generated by toggling an external switch and the signal delivered to each node explicitly. The structure of the end mission packet is given in Figure 4.8.

Destination (1 Byte)
Length (0)
Type (4)
Checksum (2 Byte)

Figure 4.8 End Mission Packet

The shutdown packet informs a peripheral that it should immediately shutdown. In normal operation this packet should not be used. It is intended to be used as an emergency shutdown of all systems. The action taken to a shutdown packet is dependent on the peripheral. For example, a motor controller should immediately turn off the motors upon receiving a shutdown packet whereas the sonar system should simply stop collecting data. Figure 4.9 gives the structure for a shutdown packet.

Destination (1 Byte)
Length (0)
Type (2)
Checksum (2 Byte)

Figure 4.9 Shutdown Packet

5.4 Broadcasting Packets

Broadcasting is supported by the AUVic-CS in order to allow the end mission and shutdown packets to be delivered to all peripherals in a timely and convenient fashion. In addition, broadcasting is directly supported by the I2C protocol making it simple to implement. The broadcast address is 0x00 and all peripherals and the hub should accept and process broadcasted packets. The hub will send a reply to the host indicating that the broadcasted packet was sent over the I2C network. However, no verification is done to ensure that all peripherals correctly received the packet.

6.0 I2C Communication

In order to understand the operation of the AUVic-CS it is necessary to understand how the I2C network operates. The I2C network is a half-duplex, master/slave network. The network is half-duplex as only one device (either a peripheral or the hub) can place data on the network at a time. In a master/slave network all communication is initiated by the master (i.e. the hub) – a slave (i.e. a peripheral) only places data onto the bus when explicitly request to do so by the master. Low-level timing and electrical details for the I2C bus can be found in [3]. Here the logical operation of the I2C network will be discussed.

Data is transferred a byte at a time and the peripheral must acknowledge each byte before operation continues. The master controls the rate data is placed on the bus by controlling the SCL line. The logic level of the SDA line must only be changed when the SCL line is low. Data is read from the network on a low to high transition of the SCL line. This allows two unique conditions to occur on the bus that are exploited by the I2C protocol. A start condition is generated by pulling the SDA line low when the SCL line is high and a stop condition is generated by allowing the SDA line to float high when the SCL line is high. The start condition signal informs all devices on the network that the master will be placing an address onto the network and the stop condition informs peripherals that the communication sequence is over and that their receive logic should be reset.

The master starts a write sequence by first generating a start condition. In the next 7 clock pulse (SCL going from low to high) the master places the address onto the bus and on the 8th clock pulse the master pulls the SDA line low to indicate a write operation (see Figure 6.1). An additional 9th clock pulse is sent by the master to allow the peripheral to acknowledge (ACK) it has received the last byte by pulling the SDA line low. Failure to ACK indicates that the address was not recognized by any peripheral on the network. Data is now sent to the peripheral 8-bits at a time followed by an acknowledge from the peripheral. The peripheral can indicate that no more data should be placed on the bus by issuing a not acknowledge (NACK) on the 9th clock pulse after a byte is read. However, in the AUVic-CS a slave never issues a NACK and communication is always terminated by the master generating a stop condition to inform the slave that no more data will be placed on the network.

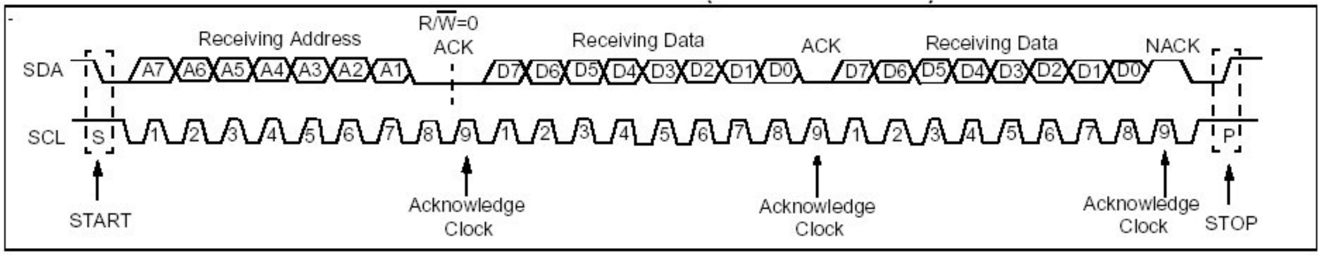


Figure 6.1 I2C Write with 7-bit Address (source: [4])

Figure 6.2 illustrates an I2C read operation. Like a write operation a start condition is generated followed by the address of the peripheral the master wishes to read data from. The 8th byte is set high to indicate a read operation. The peripheral now sends an ACK bit on the 9th clock pulse from the master by holding the SDA line low. By keeping the SDA line low the peripheral can take as much time as required to prepare the data to send to the master. Once the peripheral is ready to send a byte it lets the SDA line float high causing the master to generate clock pulses and read in the byte. On the 9th clock pulse the *master* sets the state of the SDA line and the *peripheral* latches the value. On an ACK, the slave can hold the SDA line low and prepare another byte to send. A NACK indicates that the master does not wish to read any more data from the peripheral. The communication sequence is terminated by generating the stop condition.

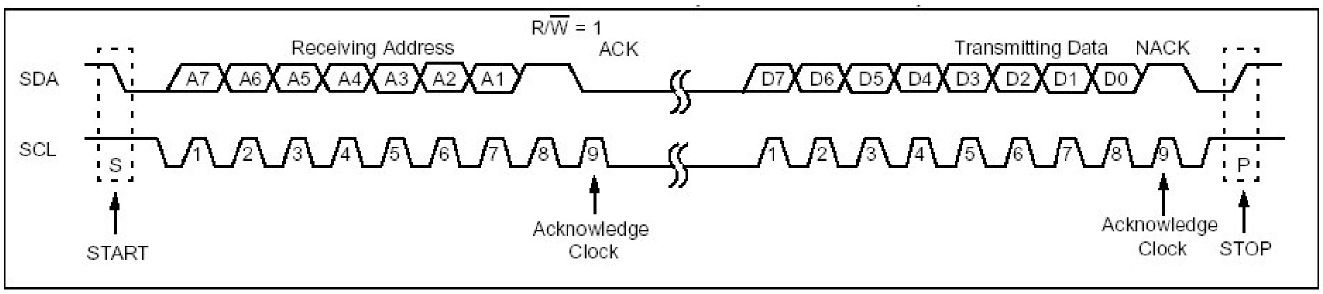


Figure 6.2 I2C Read with 7-bit Address (source: [4])

The AUVic-CS takes advantage of the nature of the I2C protocol to reduce the resource requirements of peripherals. By exploiting the I2C protocol the amount of RAM, program memory, and timers available on a peripheral can be reduced. It is assumed that peripherals are using devices that can distinguish between an address byte and a data byte in hardware (by looking for a start condition). That is, there is a flag set in hardware to indicate if the currently received byte is an address (and thus the start of a communication sequence) or a data byte. Currently, every peripheral in the AUV is using a PIC16F876 microcontroller to interface to the I2C network. The PIC16F876 meets the requirement of setting a byte in hardware to indicate the type of byte received. It is expected that the majority of microcontrollers with I2C hardware will meet this requirement. Section 7.0 discusses in detail how this is exploited to simplify the communication logic and remove the need for a communication timeout timer.

7.0 <u>Peripheral Operation</u>

Each peripheral adds functionality to AUViking that can be accessed using the AUVic-CS. The functionality provided by each peripheral and the manner in which this functionality is implemented is unique to each peripheral. In order to hide this complexity, all peripherals provide a well-defined interface to allow access to the functionality it provides.

This section explores the design specifications that every peripheral must adhere to. Failure to meet these specifications may result in the AUVic-CS failing to operate. The specifications here make no assumptions about the peripheral allowing peripherals to use a wide set of processors and electrical components.

7.1 Typical Operation

Typical operation is for the hub to transmit a packet to a peripheral and then request a response to the packet. The hub requests a response to all packets to ensure the packet was received by the peripheral without error. The peripheral receives a packets, processes it, and sets up a reply packet for the hub.

A race condition occurs between the peripheral preparing the reply packet and the hub requesting the reply. If the hub were to request a reply from a peripheral too soon it would read back an error packet. Peripherals are responsible for handling this situation by holding the SDA line low until it is ready to transmit a reply as discussed in section 6.0. This ensures the hub always reads back the intended reply from a peripheral regardless of the time it takes for a peripheral to process a packet.

Figure 7.1 gives a flow chart illustrating suitable logic for a peripheral on the AUVic-CS. The “Receiving New Packet” and “Transmit Reply” functions are contain in the I2C interrupt service routine (ISR) that is called whenever a byte is received on the I2C network. Although it is possible to built a peripheral with a processor that does not provide interrupt support for the I2C network, it is not recommended. The “Program Start” function runs whenever the ISR is not running. I2C communication will be slow relative to any microcontroller so a peripheral spends the majority of it’s processing time “Perform(ing) Slave Processing” as desired.

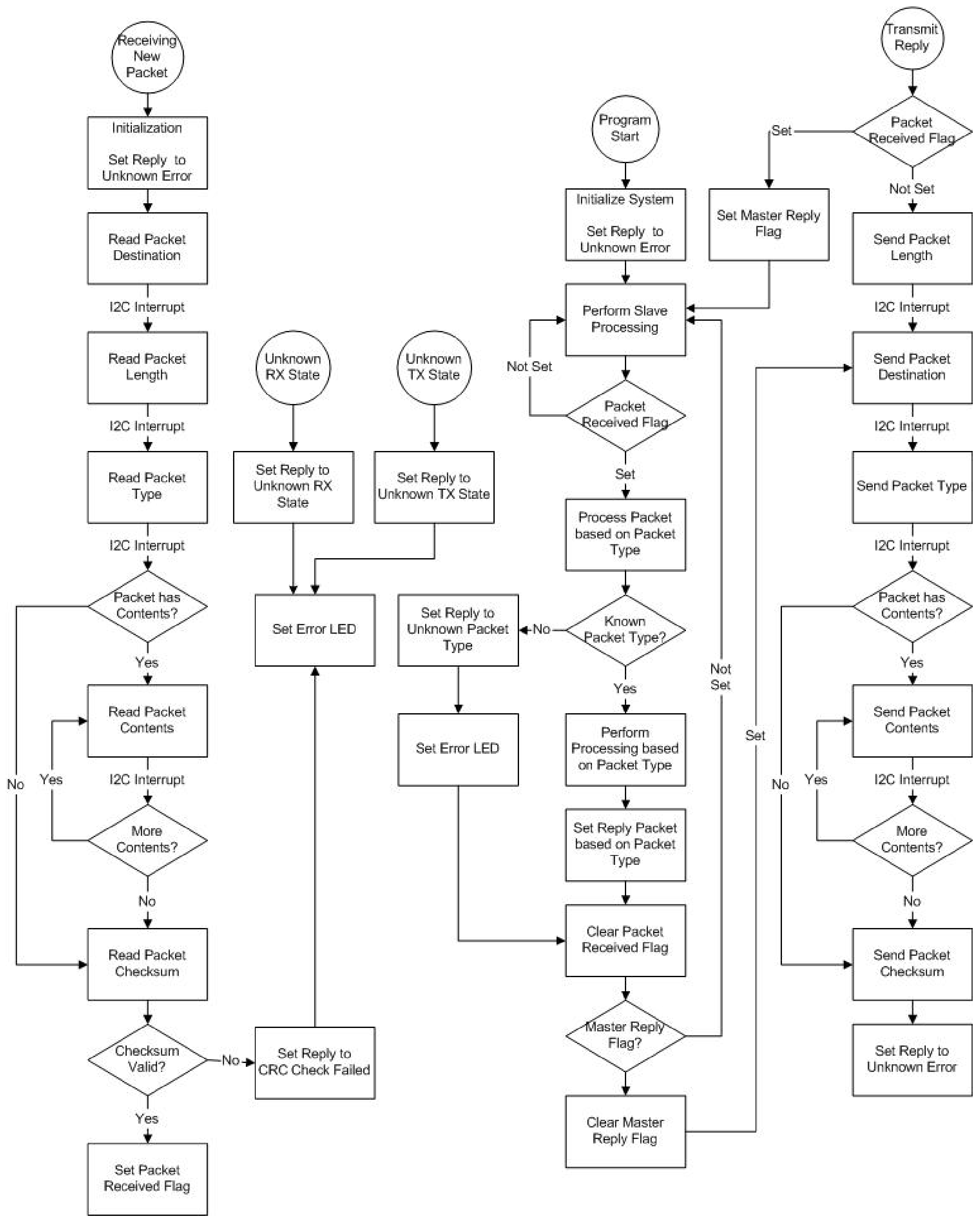


Figure 7.1 Flow Chart for Peripheral on the AUVic-CS

When a hub sends a packet to a peripheral the peripheral executes the “Receiving New Packet” logic. As discussed in section 6.0, the peripheral can determine that a new packet is to be read by checking a hardware flag and the last bit of the address byte. Once a packet is received successfully the “Packet Received” flag is set to inform the “Program Start” logic that the packet should be processed.

Transmitting a packet is handled by the “Transmit Reply” logic. The race condition discussed above is handled by using a “Master Reply” flag which informs the “Program Start” logic that the hub is pending a response and the “Packet Received” flag which informs the “Transmit Reply” logic that the last packet is still being processed and the reply packet is not ready yet.

6.2 Handling Communication Errors

The reply packet from a peripheral is always a valid reply packet. The reply packet is set to indicate an unknown error until a packet is successfully received or a specific error is detected. This ensures the hub will receive a valid error packet as a response should a communication error occur. When a valid packet is received, the reply packet is set accordingly. Once the hub reads the reply packet or starts to send a new packet to the peripheral, the reply packet is reset to an unknown error packet.

If the peripheral is able to determine the cause of a communication failure it sets the error code of the reply packet appropriately. This aids in debugging and can be used by the hub to determine how to respond to the error packet. Until a valid packet is received or an error can be detected by the peripheral, the reply packet indicates an unknown error.

Figure 6.1 indicates the communication errors that can be detected by a peripheral. An “Unknown RX State” error occurs if the hub sends a peripheral more bytes than expected and an “Unknown TX State” error occurs if the hub requests more bytes than the total length of the reply packet. A “CRC Failed” error is reported on a checksum error and an “Unknown Packet Type” error indicates that the peripheral cannot recognize the packet type.

Table 6.1 lists the errors that will be reported for different communication errors. In all cases it is assumed that the checksum is still valid. Otherwise, most errors simply result in the peripheral indicating that the checksum was invalid (the exception to this is corruption of the length byte).

Test	**Response To Hub**
Destination Corrupt – address not on I2C network	ERROR_I2C_TIMEOUT
Destination Corrupt – addressed to incorrect peripheral	Depends on Peripheral (likely an invalid packet type)
Length Corrupt – indicates additional content bytes	ERROR_SLAVE_UNKNOWN
Length Corrupt – indicates too few content bytes	ERROR_SLAVE_I2C_CRC_FAILED
Type Corrupt – indicates an unknown packet type	ERROR_SLAVE_PKT_TYPE_UNKNOWN
Type Corrupt – indicates a valid packet type	Depends on Peripheral (in general, this is a serious error)
Any Content Byte Corrupt	Depends on Peripheral and Packet Type (in general, this is a serious error)
Checksum Corrupt	ERROR_SLAVE_I2C_CRC_FAILED

Table 6.1 Response by Slave to Various Communication Errors

The I2C protocol is exploited to remove the need for a communication timeout timer. To illustrate why a communication timeout timer is required with many protocols consider the following example. Assume the length byte of a packet becomes corrupt and indicates that the length of the packet is 255 bytes (the maximum length that could be specified). The peripheral will now stay in the “Read Packet Contents” state (see Figure 5.1) until 255 bytes are received. After trying to send the packet to the peripheral several times and receiving only error packets as replies (as the peripheral is still looking for more content bytes) the hub will conclude that the peripheral is not operating correctly. The hub can now use the D1W network to reset the peripheral, but this is not a desirable way to resolve the problem as it disrupts the operation of the peripheral.

As such, it is typical to use a communication timeout timer. In this scheme when the hub detects an error packet from a peripheral it waits a set period of time before attempting communication again. This delay from the hub allows the communication timeout timer on the peripheral to expire which will cause an interrupt. During the interrupt service routine the communication logic will be reset so the peripheral is ready to receive another packet from the hub.

The use of a communication timeout timer is typical when using the RS232 protocol as it has no mechanism to indicate the start or end of a packet (the hub uses such a timer for RS232 communication with the host). The use of a communication timeout timer requires an available timer on every peripheral and a method for the hub to delay before sending a reply. This required delay by the hub also results in slower recovery from communication errors.

A communication timeout timer is not required with the I2C protocol if a hardware flag is set whenever a start condition occurs. The peripheral can now use this flag to determine if it needs to reset the communication logic. For example, if the length byte should become corrupt as illustrated above, the peripheral will recover as it resets the communication logic when a new packet (which begins with a start condition being generated) is sent from the hub. When a communication timeout timer expires it is indicating that the logic should be reset to accept a new packet – this is precisely what explicit detection of a start condition does. This allows a peripheral to recover from any communication error as the communication logic is always reset at the start of receiving or transmitting a new packet.

An important design requirement of peripherals is they must not contain state information. Removing all state information from a peripheral allows it to be reset with minimal affect on its operation. This is exploited by the host to allow recovery from otherwise fatal errors. The following two cases illustrate how state information can easily be removed from a peripheral and why this results in only minor disruption of the peripherals operation:

- *Inertial Guidance System:* The goal of the inertial guidance system (IGS) is to estimate the position of AUViking by using a 3-axis compass and an accelerometer. The current position of AUViking could be stored in the IGS, but this would result in the position state information being lost upon a reset. Instead, the IGS simply indicates the relative movement of AUViking since the last time it was polled and the host keeps track of the position of AUViking. A small lost in positional accuracy will occur after a reset, as the current relative position information in the IGS will be lost.
- *Motor Controller:* Resetting a motor controller will cause the attached motor to stop moving. The desired speed of the motor is determined by the host using data from all sensors on AUViking. This desired speed is sent to the motors at a frequency of 10Hz or greater so only a minor disruption in operation occurs.

Some peripherals may contain properties that are sent only once during initialization. For example, the motor controllers have properties specifying current and temperature limits that are set by the host only once during initialization of AUViking. *It is the responsibility of the host to resend this initialization information after it resets a peripheral.*

Resetting of a peripheral to recover from an error should only be used when no other recovery method is available. Alternative methods can often avoid the minor disruption of the peripherals operation that result from resetting. These disruptions are designed to be minimal, but are still undesirable and could hamper the operation of AUViking if they occur regularly.

8.0 <u>Hub Operation</u>

The hub is an integral component of the AUVic-CS. It is an intelligent protocol translator between the host and I2C network capable of detecting and recovering for communication errors, resetting peripherals using the D1W background network, responding to network services, and monitoring itself to determine if it is malfunctioning. All of these aspects will be examined here.

8.1 Typical Operation

In typical operation the host sends a packet to the hub that is addressed to a peripheral on the I2C network. Figure 8.1 provides a flow chart of the logic used to receive packets from the host. Packet reception is handled in an ISR. At the start of receiving a packet a communication timer is set to aid in recovering for communication errors (see section 8.2). Upon reception of the destination byte the hub determines if the packet is addressed to itself. If not, a flag is set that indicates that all bytes received should be forwarded to the I2C network. Upon reception of each byte the hub logic determines what part of the packet it is and processes the byte accordingly. When the checksum byte is received the hub sets the packet state to RECEIVED or ERROR depending on the validity of the checksum.

As shown in Figure 8.1, on reception of the destination byte a check is also made to determine if the packet should be broadcasted. A broadcasted packet requires special attention as the hub and all peripherals must process the packet and no response is required from any of the peripherals. The hub will provide the host with a response in order to meet the protocol requirements.

Figure 8.2 is a flow chart of the main processing loop of the hub. When a packet is received the state will be set to either RECEIVED or ERROR. If it is set to ERROR the hub sends an error packet to the host as a reply and than resets the communication logic. The error type will report what type of error has occurred (see section 8.2).

Otherwise, the hub checks the forward and broadcast flags to determine how to proceed. If the packet is for the hub (forward flag not set or broadcast flag set) than it is processed by the hub and an appropriate reply sent to the host. Otherwise, the hub must request a reply from the peripheral the packet was forwarded to.

Figure 8.3 is a flow chart of the logic used to request a reply from a peripheral. A communication timer is set to allow recovery from errors and than the hub attempts to read the reply. Upon successfully reading a reply it is forwarded to the host.

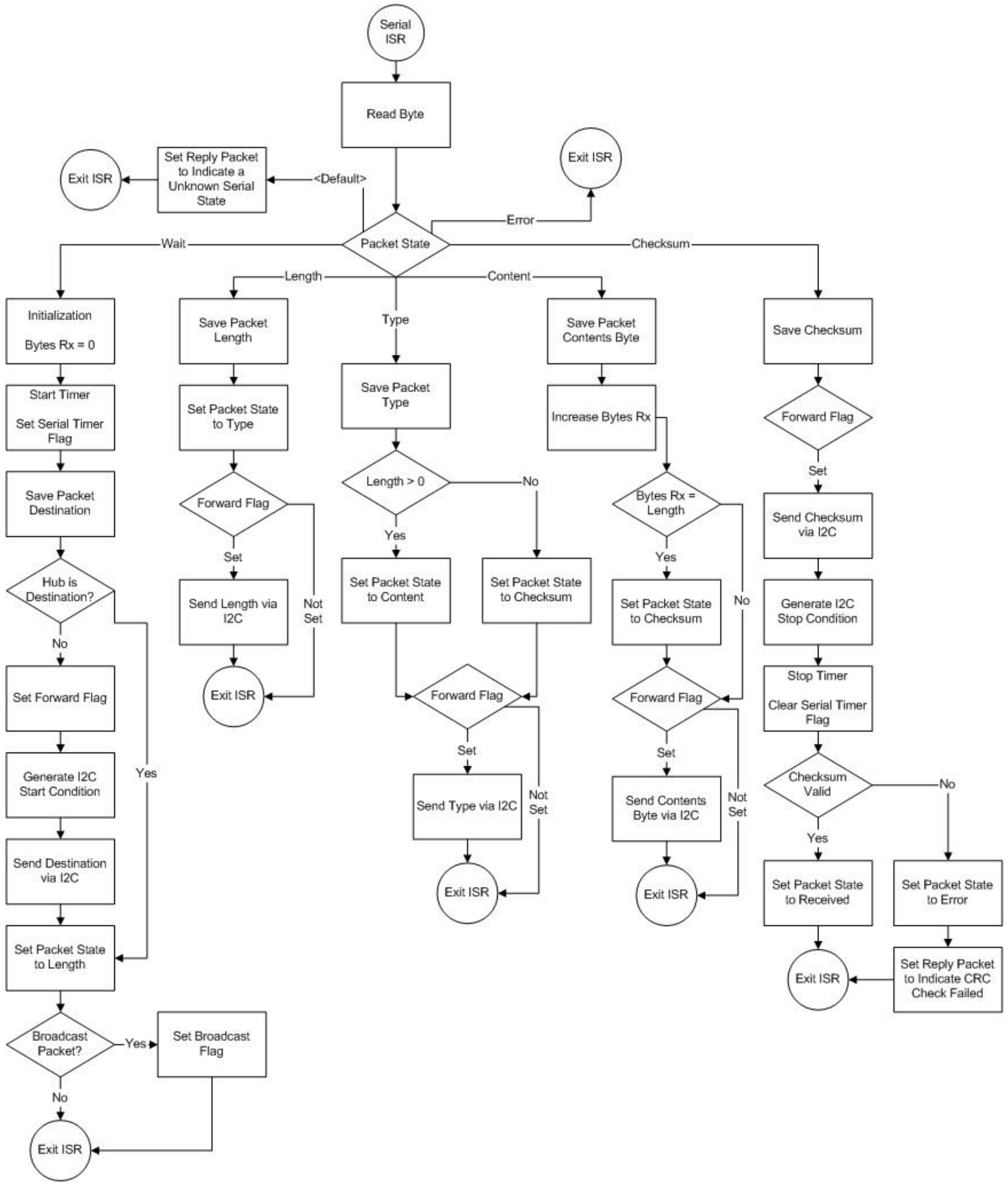


Figure 8.1 Reception and Forwarding of Packets from Host

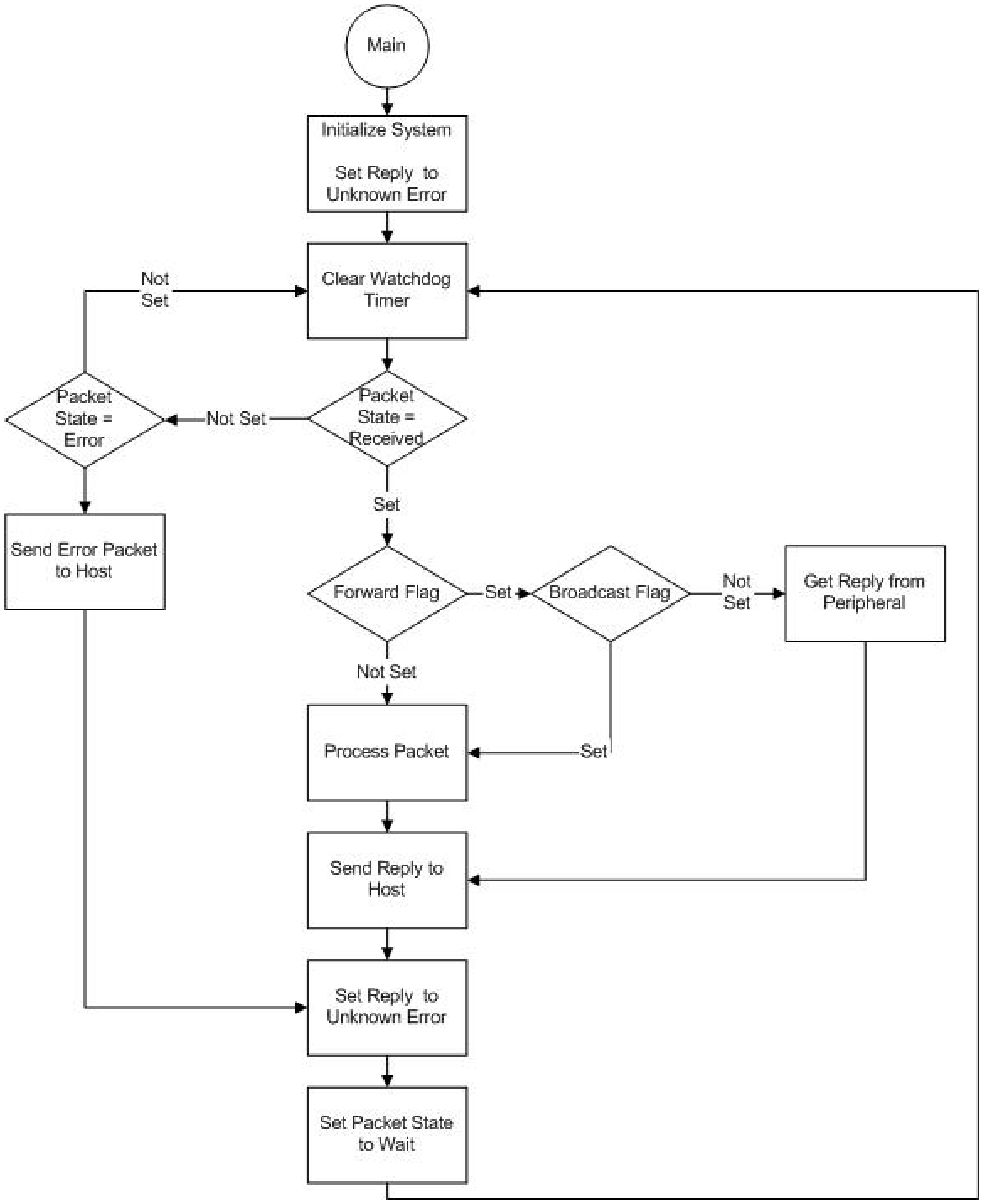


Figure 8.2 Main Processing Loop of Hub

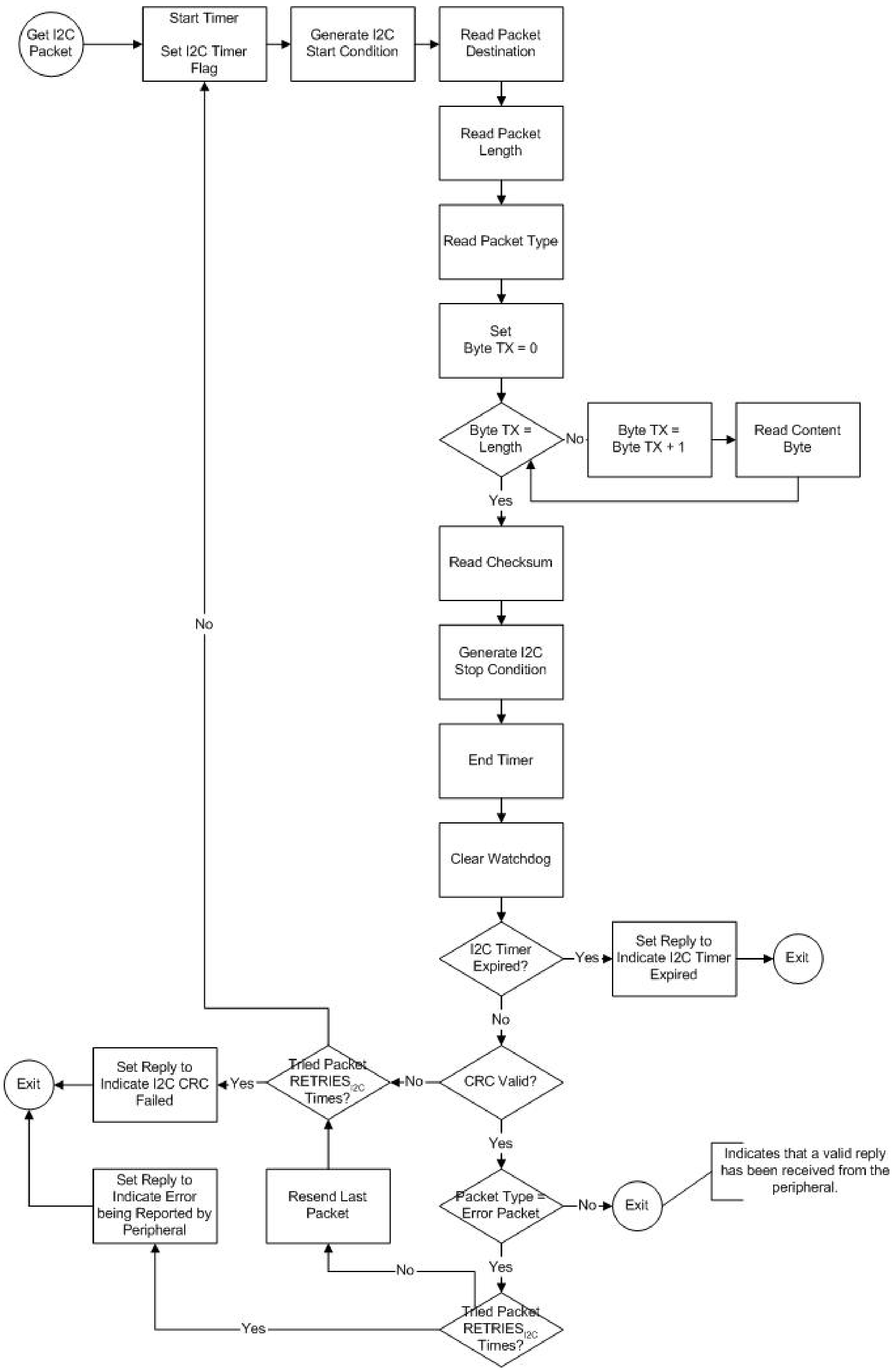

Get I2C Packet
Start Timer
Set I2C Timer Flag
Generate I2C Start Condition
Read Packet Destination
Read Packet Length
Read Packet Type
Set Byte TX = 0
Byte TX = Length
No
Byte TX = Byte TX + 1
Read Content Byte
Yes
Read Checksum
Generate I2C Stop Condition
End Timer
Clear Watchdog
I2C Timer Expired?
Yes
Set Reply to Indicate I2C Timer Expired
Exit
No
CRC Valid?
No
Tried Packet RETRIES$_{I2C}$ Times?
Yes
Set Reply to Indicate I2C CRC Failed
Exit
No
Yes
Packet Type = Error Packet
No
Exit
Indicates that a valid reply has been received from the peripheral.
Yes
Tried Packet RETRIES$_{I2C}$ Times?
No
Resend Last Packet
Yes
Set Reply to Indicate Error being Reported by Peripheral

Figure 8.3 Requesting Reply from a Peripheral

8.2 Handling Communication Errors

The hub must handle communication errors on the serial link and on the I2C network. Error detection is not performed on the D1W link. If an error occurs on the D1W network the result is the desired peripheral will not be reset. The host is free to attempt resetting a peripheral multiple times.

Communication errors on the serial link are handled by the 16-bit CRC checksum and the communication timer. Most communication errors result in one or more bits in the packet being corrupt. This results in a CRC check failure that will be reported to the host so it can re-transmit the packet if desired.

However, as illustrated in section 7.3, a communication timer is required to handle the possibility of the length byte becoming corrupt or bytes being lost. The communication timer is set when the destination byte of a packet is received. If the communication timer expires it indicates a communication error as a complete packet should have been received in this time frame. Figure 8.4 illustrates the logic of the communication timer ISR.

If the serial communication timer expires when attempting to receive a packet from the host then a reply packet is sent to the host indicating that the timer has expired. The hub now resets it's packet receive logic in preparation to receive the next packet (possibly a resend of the last packet). An I2C stop condition is also generated to inform all peripherals that the hub has finished forwarding the current packet.

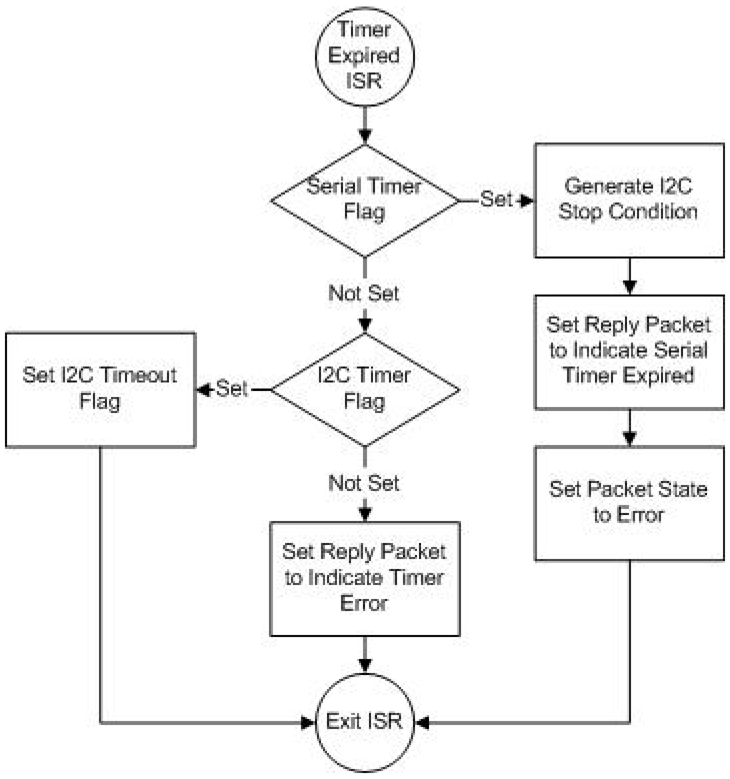

Timer Expired ISR
Serial Timer Flag
Set
Generate I2C Stop Condition
Not Set
Set Reply Packet to Indicate Serial Timer Expired
Set I2C Timeout Flag
Set
I2C Timer Flag
Not Set
Set Packet State to Error
Set Reply Packet to Indicate Timer Error
Exit ISR

Figure 8.4 Communication Timer ISR

Figure 8.3 illustrates the error recovery performed by the hub over the I2C network. A communication timer is used to ensure the hub does not wait indefinitely for a reply. If a packet is forwarded by the hub to a peripheral that does not exist on the I2C network (i.e. the destination byte is invalid or became corrupt) then the hub would wait indefinitely for a reply. When the communication timer expires (see Figure 8.4) it sets a flag that causes all I2C read operation to return immediately. The hub will then send an error packet to the host indicating the I2C timer expired.

If the timer has not expired, the hub checks to see if the checksum of the reply packet is valid and that the peripheral is not reporting an error. On an invalid checksum the hub requests the reply from the peripheral again. On an error packet the hub resends the last packet and requests the reply again. After a user defined number of attempts at obtaining a valid response the hub reports an error packet to the host. If a valid response is received it is forwarded to the host.

8.3 Communication with the Hub

Like peripherals on the I2C network the hub has an address that allows the host to send packets to it. The hub can accept diagnostic, shutdown, end mission, status, and reset packets. The effect of this network services on the hub are:

- *Diagnostic:* The host sends the hub a diagnostic packet to check if the hub is present and communicating properly. This allows the host to report an error if communication with the hub fails.
- *Shutdown:* Causes the hub to turn off all indicator LED's.
- *End Mission:* Causes the hub to turn off all indicator LED's.
- *Status:* The hub has no user programmable settings so simply returns a status reply packet with no content bytes.
- *Reset:* The host can reset any peripheral by sending a reset packet to the hub. In order for the host to reset a specific peripheral is must know the D1W address of the digital switch connected to that peripheral. This is accomplished by a simple lookup table that relates I2C addresses to D1W addresses. Figure 8.5 gives the packet structure for a reset packet.

Destination (1 Byte)
Length (0x06)
Type (0xA0)
D1W Address (6 Bytes)
Checksum (2 Byte)

Figure 8.5 Reset Packet

8.4 Watchdog Timer

A watchdog timer is used on the hub to make it more robust. When the hub is operating as desired, it will reset the watchdog timer often enough that it never expires. If the watchdog timer does expire (0.9-4.2 sec) it indicates a serious problem has occurred and the hub will reset itself. Resetting the hub should allow it to recover from any errors and allow communication between the host and I2C and D1W networks to continue.

If the watchdog timer is found to be expiring, it indicates a serious error that should be properly debugged. The watchdog timer is provided to make the hub more robust, but should not be used as a method to recover from a known error.

9.0 <u>Host Operation</u>

The host is responsible for communicating with all sensors on AUViking, processing the data from these sensors, and determining how to best set actuators on AUViking in order complete the AUVs mission. The algorithms used to process sensor information and follow mission plans is critical to AUViking, but have no influence over the AUVic-CS so will not be discussed. This section will focus on how the host interacts with the communication system.

9.1 Round-robin Schedule

A straight forward, round-robin schedule is used on AUViking in order to obtain sensor information and set actuators at a roughly periodic rate. A round-robin schedule operates by simply sending packets in the order they are listed in the schedule and wrapping around to the start of the schedule after the last packet in the schedule is sent. The round-robin schedule is simple to implement and the desired 10Hz packet frequency can easily be obtained. Its limitation is that it has no functionality to send some packets more frequently than others.

An alternative scheduling system could be used that allowed more flexibility over how often a packet was sent. For example, a timer could be used to only send the battery voltage packet only once a second as this sensor information is not required at 10Hz or a priority scheme could be used that indicates the relative frequency packets should be sent. The round-robin schedule can simulate relative priorities by scheduling some packets more often than others. However, the high data rate of the AUVic-CS allows these more complicated scheduling systems to be avoided as there is no harm in receiving packets more often than required as long as all packets are sent at the desired frequency.

A round-robin schedule that could be used on AUViking is shown in Figure 9.1. Many of these packets (marked with a * in Figure 9.1) are not required by the algorithms on AUViking, but have been added as they provide extra information about the state of AUViking that may provide useful information when testing AUViking.

3 * Set Motor
3 * Get Motor Flags
3 * Get Motor Status*
Get Pressure Data
Get Battery Voltages
Get Temperature Flags
Get Sensor Status*
Get Sonar Data
Get Sensor Status*
Get Inertial Guidance Data
Get Compass Data
Get IGS Status*

Figure 7.1 Round-robin Schedule

9.2 Typical Operation

A simplified, high level overview of how the host operates is given in Figure 9.2. When AUViking first comes online it initializes the system (turns off LED's, clears system variables, etc.) and makes sure the AUVic-CS is operating correctly by sending a diagnostic packet to the hub and all peripherals. If an error results from any of the diagnostic packets all systems are sent the diagnostic packet again until either no errors are reported or the AUVic-CS is deemed to be malfunctioning. Ideally, AUViking would attempt to determine what systems are online and try to complete the mission with only these systems. In practice, this is time consuming to implement and it is easier to simply have the host report an error so the problem can be resolved.

From the point of view of the AUVic-CS the operation of the host is trivial. It simply sends all packets using the round-robin schedule followed by processing the received data. The result of processing the data is how to set the motor speed and directions to best complete the current goal of AUViking. This information is sent to the motors with the *set motor* packet that is at the top of the round-robin schedule.

This process is repeated until the mission is completed or aborted. A mission may be aborted either by the host determining the maximum mission time has elapsed or an external switch being toggled. At this time all systems are sent an end mission packet and the AUV is brought to the surface by slowly turning off the motors.

9.3 Handling Communication Errors

Communication errors will occur and it is critical that the host be able to respond correctly to any communication error it might receive. There are two possible sources of errors that host must be able to detect: a lost packet on the RS232 link and an error packet as a reply. As shown in Figure 9.2, a communication timer is used to ensure a response is received in a reasonable amount of time (nominal 6ms) in order to detect lost packets and all responses are checked to determine if a communication error is being reported.

A valid response to any communication error is to resend the packet. This is due to the fact that the most likely cause of a communication error is noise in the environment corrupting the packet. Because of this, the host will always resend a packet three times before assuming further action is required.

After receiving three consecutive errors the host will report the error via an LED and to the dockside PC via TCP/IP (if the TCP/IP channel is available). This situation represents a serious error that indicates the environment is too electrically noisy or a logical error exists in one of the systems (for example, a peripheral has entered a logical state that is causing it to continually calculate the checksum incorrectly).

The action taken by the host is determined by the *last* communication error it receives as it assumes all three communication errors were the same to simplify the error handling logic. This is a reasonable assumption as logical errors will most likely result in the same communication error whereas an electrically noisy environment will result in random errors. Since there is no action the host can take to improve the level of electrical noise in the environment the host simply assumes the error is a logical one and that all three errors were the same.

Given that proper testing is done on each system before and after it is integrated into AUViking it is reasonable to assume that any logical errors that appear in a system (host, hub, or peripheral) will be due to a rarely occurring event (i.e. a specific communication error, a unique sequence of packets). Recovering from such an error is difficult as the system has likely entered a state it was not intended to enter. Even if the logical error can be determined it is highly probable the host will not be able to resolve the problem by simply sending it a certain sequence of packets.

The hub and all peripherals have been designed so they can be reset without seriously affecting their operation. This design requirement allows a peripheral to be reset at any time in order to attempt recovery from unforeseen errors. The host uses this to attempt recovery once three communication errors have been received. A check is made to determine if the error is being reported by the hub or a peripheral. If the hub is reporting the error the host will wait for 5 seconds in order to allow the watchdog timer on the hub to expire. This will reset the hub and hopefully resolve the error. For peripherals, the D1W reset network is used to reset the peripheral in hopes of resolving the error. In either case, the host assumes the error is resolved and simply continues on to the next packet in the round-robin schedule.

There are several situations where this error recovery mechanism will fail. As an example, if the hub were in a state where it is clearing the watchdog timer, but failing to respond to packets then the AUVic-CS will cease to operate. No communication system is without the possibility of failure which is why many commercial systems contain redundant communication channels and backup systems. For financial reasons this is not feasible on AUViking and the nature of AUViking makes it reasonable to simply perform a system wide reset should the current recovery mechanisms fail.

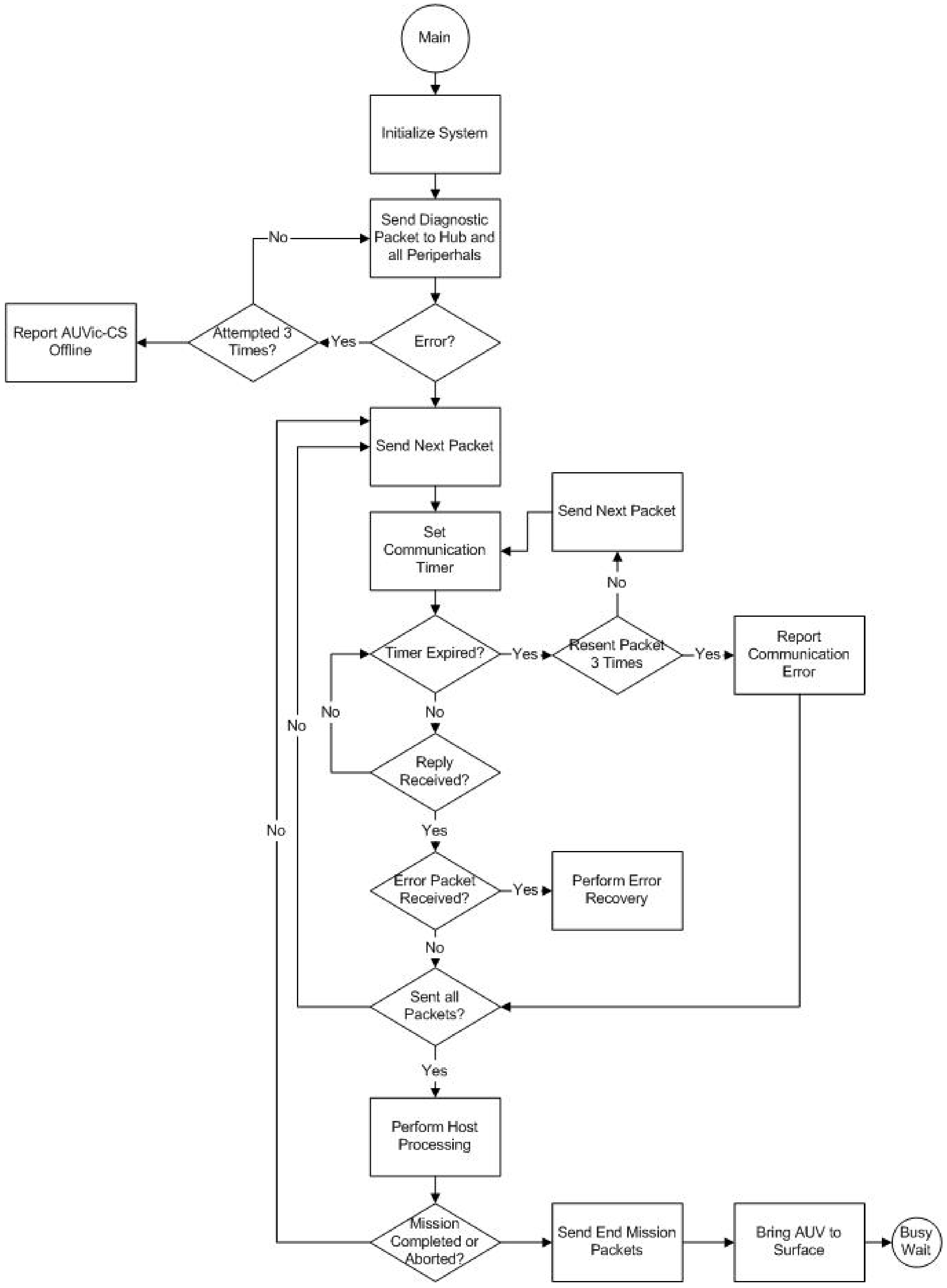


Figure 9.2 Main Processing Loop of Host

10.0 Programming

The PIC16F876 on the hub can be programmed either by in-circuit serial programming (ICSP) or by the resident bootloader via the serial port.

To program the hub using ICSP requires a PIC programmer. The circuit has been designed for compatibility with the Newfoundland Warp13 programmer (www.newfoundelectronics.com), but should be compatible with all programmers supporting ICSP. ICSP is intended to be used only when loading the PIC with a new bootloader.

Programming the PIC using a bootloader does not require a PIC programmer and allows new programs to be downloaded much quicker. Because of this, the motor controller program has been designed to ensure there is sufficient memory free to allow a bootloader. The motor controller uses Shawn Tolmie's bootloader (www.microchipc.com). Complete instruction on how to use the bootloader can be found on his website.

11.0 Conclusion

The AUVic Communication System allows the host to communicate with all systems in the AUV. It is hoped the AUVic-CS will provide robust enough to find utility in future iterations of the AUVic AUV. Future expansion is support as the I2C and D1W networks can both support multiple slaves and the baudrate of all communication links can be increased.

Cited References

[1] ESAcademy, “The I2C Bus Q&A Section”, date unknown, [cited 2002 December 6] Available at: http://www.esacademy.com/faq/i2c/q_and_a/i2cqena.htm

[2] Williams, Ross. “A Painless Guide To CRC Error Detection Algorithms”, 1992 August 19, [cited 2002 December 5], Available at: http://www.piclist.com/techref/method/math/crcguide.html

[3] Phillips Semiconductor, “The I2C Bus Specification v2.0”, 1998 December, [cited 2002 December 7], Available at: http://www.semiconductors.philips.com/acrobat/various/I2C_BUS_SPECIFICATION_2.pdf

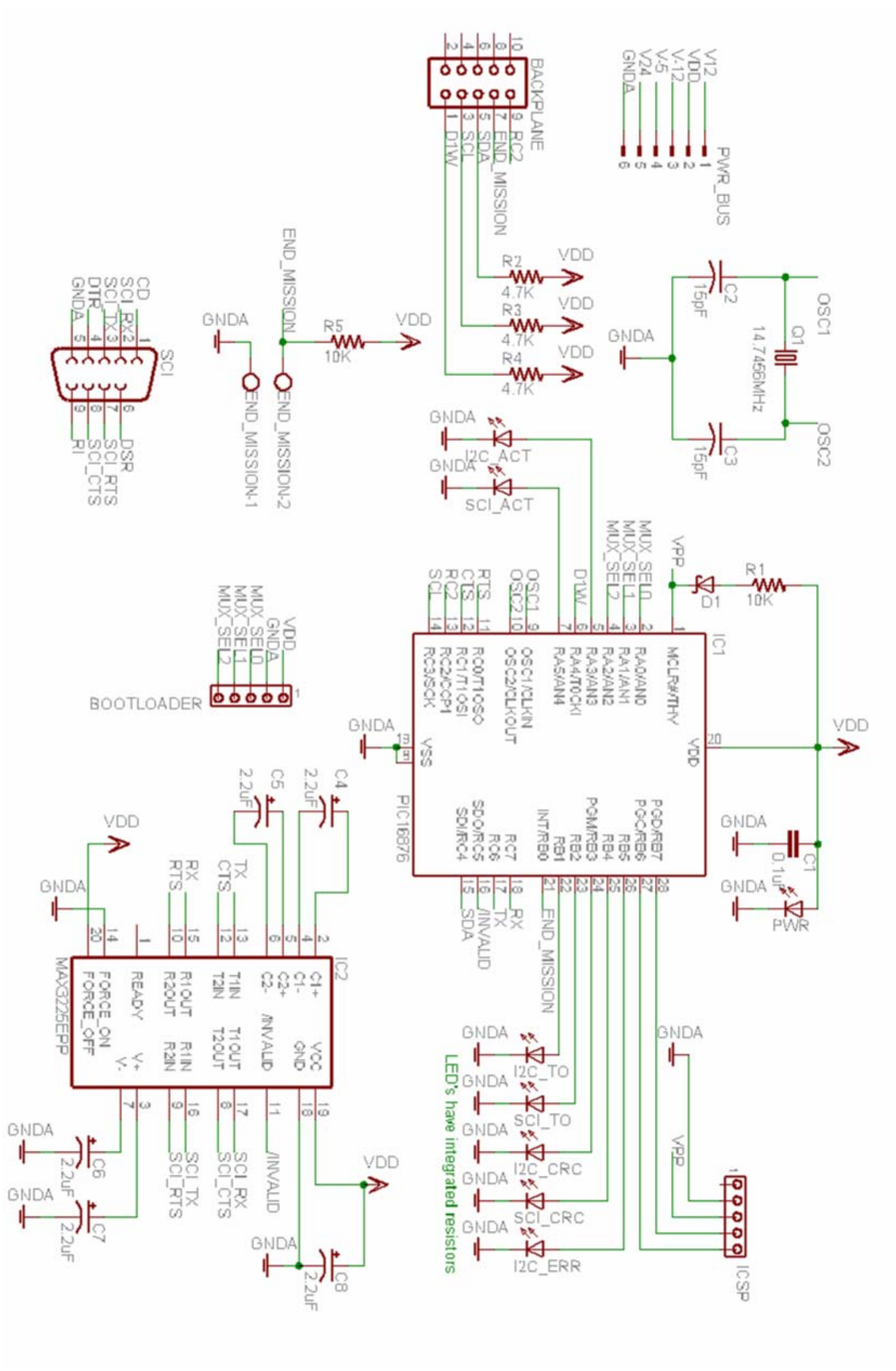
[4] Bowling, Stephen (Microchip). “Using the PICmicro SSP for Slave I2C Communication”, 2000, [cited 2002 December 8], Available at:

General References

Microchip, “PIC16F87x Data Sheet”, date unknown, [cited 2002 December 3], Available at: http://www.microchip.com/download/lit/pline/picmicro/families/16f87x/30292c.pdf

Maxim, “Dual Addressable Switch Plus 1-kbit Memory Datasheet”, date unknown, [cited 2002 December 6], Available at: http://pdfserv.maxim-ic.com/arpdf/DS2406.pdf

Appendix I – Hub Schematic

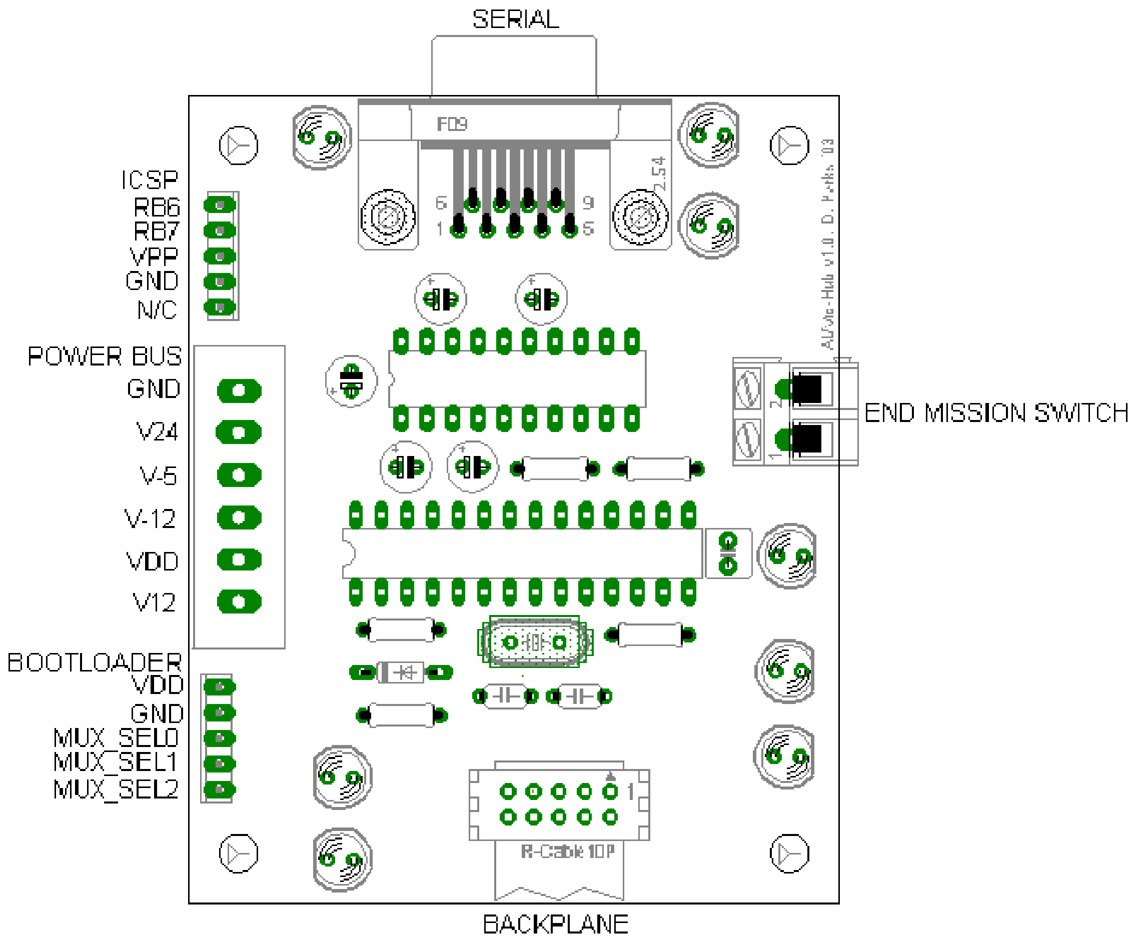


Appendix II – Bill of Material

Count	Component Name	Location	Value	Description	Digi-key #	Price Each*	Total Cost
1	PIC16F876-20/SP	IC1	uC	Microchip PIC16F876 20MHz PDIP Microcontroller	PIC16F876-20/SP-ND	13.01	13.01
1	MAX3225EPP	IC2	IC	1Mbps RS-232 TRANSCEIVER	N/A	sample	0.00
1	ECS-147.4-S-4	Q1	XTAL	14.7456 MHZ QTZ CRYSTAL	X431-ND	1.33	1.33
1	C320C104M5U5CA	C1	0.1uF	CAP 50V 20% CER RADIAL	399-2155-ND	0.25	0.25
2	ECU-S2A150JCA	C2, C3	15pF	100V MONOLITH CERM CAP	P4839-ND	0.63	1.26
5	ECE-A1EKK2R2	C4-C8	2.2uF	CAP ELECT 25V KK RADIAL	P971-ND	0.34	1.70
1	1N5817-T	D1	DIODE	DIODE SCHOTTKY 20V 1A	1N5817DICT	0.65	0.65
1	CFR-25JB-10K	R1	10K	1/4W 5% CARBON FILM RES	10KQBK-ND	0.092	0.10
3	CFR-12JB-4K7	R2-R4	4.7K	1/8W 5% CARBON FILM RES	4.7KEBK-ND	0.092	0.28
7	LTL-4233-R1	*	LED	LED RES LAMP 5MM 5V	160-1051-ND	0.43	3.01
1	LTL-4223-R2	PWR	LED	LED RES LAMP 5MM 12V RED	160-1112-ND	0.43	0.43
2	22-05-2051	ISCP, BOOTLOADER	CONN	5 CIR HED.100 FRICT LOCK	WM4203-ND	1.30	2.60
		BACKPLANE	CONN				
		END_MISSION	CONN				
		POWER_BUS	CONN				
1	182-009-212-161	SCI	CONN	DB9 FMAL R/A SHELL .318	182-709F-ND	1.82	1.82

* I2C_ACT, SCI_ACT, I2C_TO, SCI_TO, I2C_CRC, SCI_CRC, and I2C_ERR (Note: LED's have built in resistors)

Appendix III – PCB Layout



Appendix IV – PIC16F876 Pinout

Pin	Pin Name	Sonar Module Function	Sub-System
1	*MCLR/V_{PP}	Connected to ISP header	ISP
2	RA0/AN0	Bootloader MUX Select 0	MUX Select
3	RA1/AN1	Bootloader MUX Select 1	MUX Select
4	RA2/AN2/V_{ref}-	Bootloader MUX Select 2	MUX Select
5	RA3/AN3/V_{ref}+	I2C Activity LED	Indicator LED
6	RA4/TOCKI	Dallas 1-wire Bus	Dallas 1-wire
7	RA5/AN4/*SS	Serial Activity LED	Indicator LED
8	V_{SS}	Ground	Power
9	OSC1/CLKIN	Connected to 14.7456MHz oscillator	Timer
10	OSC2/CLKOUT	Connected to 14.7456MHz oscillator	Timer
11	RC0/T10S0/T1CKI	Serial Communication RTS Line	Serial
12	RC1/T10S1/CCP2	Serial Communication CTS Line	Serial
13	RC2/CCP1	Spare	Spare*
14	RC3/SCK/SCL	I2C Communication Clock Line	I2C
15	RC4/SDI/SDA	I2C Communication Data Line	I2C
16	RC5/SDO	Serial Invalid Pin	Serial
17	RC6/TX/CK	Serial Communication Tx Line	Serial
18	RC7/RX/DT	Serial Communication Rx Line	Serial
19	V_{SS}	Ground	Power
20	V_{DD}	Regulated +5V	Power
21	RB0/INT	Shutdown Signal	Shutdown
22	RB1	I2C Timeout LED	Indicator LED
23	RB2	SCI Timeout LED	Indicator LED
24	RB3/PGM	I2C CRC Failed LED	Indicator LED
25	RB4	Serial CRC Failed LED	Indicator LED
26	RB5	System Lost LED	Indicator LED
27	RB6/PGC	Connected to ISP header	ISP
28	RB7/PGD	Connected to ISP header	ISP

* Spare pins are brought are connected to the network header for future expansion.

Appendix V – Error Codes

Error Code	Error Name	Purpose
0xE0	ERROR_SCI_CRC_FAILED	Indicates hub has detected a CRC error with the last packet received from the host.
0xE1	ERROR_I2C_CRC_FAILED	Indicates the hub has detected a CRC error with the last packet received from a peripheral.
0xE2	ERROR_PKT_TYPE_UNKNOWN	Indicates the hub has received a packet from the host with a packet type it does not know how to process.
0xE3	ERROR_I2C_RX_STATE_UNKNOWN	Indicates the hub has entered an invalid state when trying to receive a packet from a peripheral.
0xE4	ERROR_I2C_TX_STATE_UNKNOWN	Indicates the hub has entered an invalid state when trying to send a packet to a peripheral.
0xE5	ERROR_SCI_STATE_UNKNOWN	Indicates the hub has entered an invalid state when trying to receive or send a packet using the serial port.
0xE6	ERROR_SCI_TIMEOUT	Indicates the hub has timed out while waiting to receive a packet from the host.
0xE7	ERROR_I2C_TIMEOUT	Indicates the hub has timed out while waiting to receive a packet from a peripheral.
0xE8	ERROR_I2C_MAXIMUM_RETRIES	Indicates the hub has tried to resend a packet to a peripheral a user defined number of times and was unable to deliver it successfully.
0xE9	ERROR_TIMER	Indicates the hub communication timer has expired, but the reason is unknown.
0xEF	ERROR_UNKNOWN	Indicates the hub has encountered a problem receiving the last packet from the host, but the nature of the problem in unknown.
0xF0	ERROR_SLAVE_SCI_CRC_FAILED	Indicates a peripheral has detected a CRC error with the last packet received from the serial port (some peripherals can communicate using I2C or the serial port for debugging purposes).
0xF1	ERROR_SLAVE_I2C_CRC_FAILED	Indicates a peripheral has detected a CRC error with the last packet received from the hub.
0xF2	ERROR_SLAVE_PKT_TYPE_UNKNOWN	Indicates a peripheral has received a packet from the hub with a packet type it does not know how to process.
0xF3	ERROR_SLAVE_I2C_RX_STATE_UNKNOWN	Indicates a peripheral has entered an invalid state when trying to receive a packet from the hub.
0xF4	ERROR_SLAVE_I2C_TX_STATE_UNKNOWN	Indicates a peripheral has entered an invalid state when trying to send a packet to the hub.
0xF5	ERROR_SLAVE_SCI_STATE_UNKNOWN	Indicates a peripheral has entered an invalid state when trying to receive or send a packet using the serial port.
0xFF	ERROR_SLAVE_UNKNOWN	Indicates the slave has encountered a problem receiving the last packet from the hub, but the nature of the problem in unknown.